1. **[ALREADY GIVEN] Write a parallel or distributed program that counts the number of permutations of `N` that satisfy a given property. You have a function (`bool pred(vector <int> const& v)`) that verifies if a given permutation satisfies the property. Your program shall call that function once for each permutation and count the number of times it returns true.**

```
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>

using namespace std;

bool check(vector <int> v) {
  if(v[0] % 2 == 0) {
    return true;
  }
  return false;
}

bool contains(vector <int> v, int n) {
  for(auto it : v) {
    if(it == n) {
      return true;
    }
  }
  return false;
}

atomic <int> cnt;

void back(vector <int> sol, int T, int n) {
  if(sol.size() == n) {
    if(check(sol)) {
      cnt ++;
    }
    return;
  }
  if(T == 1) {
    for (int i = 1; i <= n; ++ i) {
      if(contains(sol, i)) continue;
      sol.push_back(i);
      back(sol, T, n);
      sol.pop_back();
    }
  } else {
    vector <int> x(sol);
    thread t([&](){
      for(int i = 1; i <= n; i += 2) {
        if(contains(x, i)) continue;
        x.push_back(i);
        back(x, T / 2, n);
        x.pop_back();
      }
    });
    for(int i = 2; i <= n; i += 2) {
      if(contains(sol, i)) continue;
      sol.push_back(i);
      back(sol, T - T / 2, n);
```

```
      sol.pop_back();
    }
    t.join();
  }
}

int main() {
  back(vector <int>(), 2, 3);
  cout << cnt.load() << '\n';
}
```

1. **Write a parallel (distributed or local, at your choice) program for finding a Hamiltonian path starting at a given vertex. That is, you are given a graph with n vertices and must find a path that starts at vertex 0 and goes through each of the other vertices exactly once. Find a solution, if one exits. If needed, assume you have a function that gets a vector containing a permutation of length n and verifies if it is Hamiltonian path in the given graph or not.**

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>

using namespace std;

const int maxn = 1005;

int n, m;

bool solved = false;
vector <int> sol;
vector <int> g[maxn];

inline void back(vector <int> nodes, int T) {
  if(solved) {
    return ;
  }
  if(nodes.size() == n) {
    if(!solved) {
      solved = true;
      sol = nodes;
    }
    return ;
  }
  if(T > 1) {
    vector <int> adj = g[nodes.back()];
    thread t([adj, nodes, T](){
      cerr << "new thread\n";
      for(int i = 0; i < adj.size(); i += 2) {
        if(find(nodes.begin(), nodes.end(), adj[i]) == nodes.end()) {
          vector <int> aux(nodes);
          aux.push_back(adj[i]);
          back(aux, T / 2);
        }
      }
    });
```

```
    for(int i = 1; i < adj.size(); i += 2) {
      if(find(nodes.begin(), nodes.end(), adj[i]) == nodes.end()) {
        vector <int> aux(nodes);
        aux.push_back(adj[i]);
        back(aux, T - T / 2);
      }
    }
    t.join();
  } else {
    for(auto it : g[nodes.back()]) {
      if(find(nodes.begin(), nodes.end(), it) == nodes.end()) {
        vector <int> aux(nodes);
        aux.push_back(it);
        back(aux, T);
      }
    }
  }
}

int main() {
  ifstream fin("input.in");
  ofstream fout("output.out");
  fin >> n >> m;

  for(int i = 0; i < m; ++ i) {
    int x, y;
    fin >> x >> y;
    g[x].push_back(y);
    g[y].push_back(x);
  }
  vector <int> init;
  init.push_back(1);
  back(init, 6);
  if(!solved) {
    cout << "Not found!\n";
  } else {
    for(auto it : sol) {
      cout << it << '\n';
      fout << it << '\n';
    }
  }
}
```

2. **[ALREADY GIVEN] Consider the following code for transferring money from one account to another. You are required to write a function parsing all accounts (assume you have a `vector <Account>`) and compute the total amount of money there, so that it doesn't interfere with possible transfers at the same time. Change the transfer function if needed, but it must be able to be called concurrently for independent pair of accounts.**

```
struct Account {
    unsigned id;
    unsigned balance;
    mutex mtx;
};

bool transfer(Account& from, Account& to, unsigned amount) {
  {
    unique_lock<mutex> lck1(from.mtx);
    if(from.balance < amount) return false
    from.balance -= amount;
  }
  {
    unique_lock<mutex> lck2(to.mtx);
    to.balance += amount;
  }
}
```

Concurrentcy issue

There is a problem with this method if we want to compute the sum, because it may be the case that the amount is taken out of an account, but not yet added to the other one, such that some amount is lost from the entire sum.

Solution

Lock the two nodes simulateously

Code

```
bool transfer(Account& from, Account& to, unsigned amount) {
  unique_lock<mutex> lck1(from.mtx);
  unique_lock<mutex> lck2(to.mtx);
  if(from.balance < amount) return false
  from.balance -= amount;
  to.balance += amount;
  return true;
}

int getSum(vector <Account> v) {
  int sum = 0;
  for(auto account : v) {
    account.mtx.lock();
    sum += account.balance;
  }
  for(auto account : v) {
    account.mtx.unlock();
  }
  return sum;
}
```

3. We have n servers that can communicate with each other

> There are events producing on each of them; each event has an associated information. Each server must write a history of all events (server and event info) produced on all servers, and, furthermore, the recorded history must be the same for all servers. Write a distributed algorithm that accomplishes that. Consider the case n = 2 for starting.

Solution

Let's consider the case when `n = 2`. Let's name the processes `A` and `B`.

Every process will keep a `Lamport clock` and pass that on each message.

Suppose an event occurs in the process `A`. Then, process `A` will increate it's timestamp `t` by one, and send that event alongside `t` at the process `B` - this is called `PREPARE` event. Process `B` computes the maximum between its internal clock and the timestamp of the message and adds one. It send back an `OK` with that computed timestamp. Process `A` receives the `OK` and sends back a `COMMIT` message. They both agreed on this value.

The generalization to `n > 2` follows easily.

Each process having an occuring event will:

- broadcast `PREPARE` with the timestamp
- waits for `OKs` from all the other processes
- computes the maximum amongs the `OKs` timestamps
- broadcast `COMMIT` to each process

There is only one little problem. A process may have previously gave an `OK` but did not get any `COMMIT` yet for that event. So, in case another `COMMIT` appears, it can't write that to a file because the `COMMIT` from the previously sent `OK` may be either before, or after the current `COMMIT`. That's why, each process will maintain a list of given `OKs` as well as a list of `COMMITs` to be flushed to disk.

Note. Every tie can be solved by chosing an initial *arbitrary* ordering of the processes. Such as the `PID` or `IP` if they live on different hosts.

1. **Write a parallel (distributed or local, at your choice) that computes the discrete convultion of a vector with another vector. The convolution is defined as: `r(i) = sum(a(i) * b(i – j), j = 0..n)`**

```
#include <iostream>
#include <vector>
#include <thread>

using namespace std;

inline void solve(vector <int> a, vector <int> b, int T) {
  vector <thread> threads;
  int n = a.size();
  vector <int> sol(n, 0);
  for(int idx = 0; idx < T; ++ idx) {
    threads.push_back(thread([a, b, idx, n, &sol, T](){
      for(int i = idx; i < n; i += T) {
        for(int j = 0; j < n; ++ j) {
          sol[i] += a[j] * b[(i - j + n) % n];
        }
      }
    }));
  }
  for(int i = 0; i < threads.size(); ++ i) {
    threads[i].join();
  }
  for(auto it : sol) {
    cerr << it << '\n';
  }
}

int main() {
  solve({1, 2, 3}, {1, 2, 3}, 3);
  // r[0] = a[0] * b[0] + a[1] * b[2] + a[2] * b[1] = 1 * 1 + 2 * 3 + 3 * 2 = 1
+ 6 + 6 = 13
  // r[1] = a[0] * b[1] + a[1] * b[0] + a[2] * b[2] = 1 * 2 + 2 * 1 + 3 * 3 = 2
+ 2 + 9 = 13
  // r[2] = a[0] * b[2] + a[1] * b[1] + a[2] * b[0] = 1 * 3 + 2 * 2 + 3 * 1 = 3
+ 4 + 3 = 10
}
```

2. **[ALREADY GIVEN] Consider the following code for enqueueing a continuation on a future. Identify and fix the thread-safety issue.**

```
template<typename T>
class Future {
  list<function<void(T)>> continuations;
  T val;
  bool has_value;
public:
  Future(): has_value(false) {}
  void set(T v) {
    val = v;
    hasValue = true;

    for(function<void(T)>& f: continuations) {
      f(v);
    }

    continuations.clear();
  }
  void addContinuation(function<void(T)> f) {
    if(hasValue) {
      f(val);
    } else {
      continuations.push_back(f);
    }
  }
}
```

Concurrentcy issue

There is no synchronization mechanism on the hasValue variable which is crucial to the algorithm.

Solution

Create a mutex on the hasValue function, assuming that the iteration on a `list` is thread safe. Otherwise, simply add a mutex so that only one method can be called at a time.

3. [ALREADY GIVEN] Write a parallel algorithm that computes the product of 2 matrices.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>

using namespace std;

int n;
vector <vector <int>> a, b, c;

inline void solve(int T) {
  vector <thread> t;
  for(int idx = 0; idx < T; ++ idx) {
    t.push_back(thread([&, idx, T](){
      for(int i = idx; i < n; i += T) {
        for(int j = 0; j < n; ++ j) {
          for(int k = 0; k < n; ++ k) {
            c[i][j] += a[i][k] * b[k][j];
          }
        }
      }
    }));
  }
  for(int i = 0; i < t.size(); ++ i) {
    t[i].join();
  }
}

int main() {
  ifstream fin("input.in");

  fin >> n;

  for(int i = 0; i < n; ++ i) {
    a.push_back(vector <int> ());
    c.push_back(vector <int> ());
    for(int j = 0; j < n; ++ j) {
      int x;
      fin >> x;
      a.back().push_back(x);
      c.back().push_back(0);
    }
  }
  for(int i = 0; i < n; ++ i) {
    b.push_back(vector <int>() );
    for(int j = 0; j < n; ++ j) {
      int x;
      fin >> x;
      b.back().push_back(x);
    }
  }


  solve(5);

  for(int i = 0; i < n; ++ i) {
    for(int j = 0; j < n; ++ j) {
```

```
      cerr << c[i][j] << ' ';
    }
    cerr << '\n';
  }
}
```